Truly Nested Data-Parallelism: Compiling SaC for the MicroGrid Architecture*

Stephan Herhut[1], Carl Joslin[1], Sven-Bodo Scholz[1], and Clemens Grelck[1,2]

[1] University of Hertfordshire, School of Computer Science
Hatfield, Herts, AL10 9AB, United Kingdom
{s.a.herhut,c.a.joslin,s.scholz,c.grelck}@herts.ac.uk
[2] University of Amsterdam, Institute of Informatics
Science Park 107, 1098 XG Amsterdam, Netherlands
c.grelck@uva.nl

Abstract. Data-parallel programming facilitates elegant specification of concurrency. However, the composability of data-parallel operations so far has been constrained by the requirement to have only flat data-parallel operation at runtime. In this paper, we present early results on our work to exploit hardware support for nested concurrency to directly map nested data-parallel operations in high-level specifications to low-level codes that can be efficiently executed. To this effect, we have devised a compilation scheme from data-parallel operations in SaC to the systems programming language of the Microgrid architecture. Furthermore, we present early empirical results to assert the viability of our approach.

1 Introduction

With the advent of multi-core processors in the mainstream, the question of how to efficiently program concurrent systems has gained more attention again. Whereas previously existing software profited automatically from advances in processor design, this is no longer the case with the current trend towards parallel architectures. Instead, software needs to be explicitly adapted to benefit from the increased processing power through concurrent execution. However, explicitly encoding concurrency by means of low-level language constructs is cumbersome. Firstly, factoring in concurrency makes the already complex art of programming even more difficult. And secondly, manual concurrency engineering not only is error-prone but the resulting bugs are hard to catch.

Thus, a different approach for programming in a parallel setting is required that abstracts from the low-level details and allows the programmer to concentrate on the inherent parallelism of an algorithm. Here, the data-parallel approach towards programming comes to mind. In data-parallel programming, a, usually uniform, operation is applied to each element of a collection of data. In case of a *parallel map*, the result then is a corresponding collection where each

* This work was funded by the European Union Apple-CORE project grant no. FP7 215216.

element is computed by applying the operation to the corresponding element of the source collection. For *parallel folds*, a single element is computed by reducing all elements in the collection using the given operation.

A common scenario in data-parallel computing are operations on arrays, e.g., computing the element-wise addition or summing up all elements of an array. Early incarnations of programming languages that support a data-parallel programming style on arrays are HPF [1] and SISAL [2]. The former, *High-Performance Fortran*, supports data-parallel intrinsics for array operations and has a special `FORALL` construct for data-parallel loops in general. The latter, SISAL, allows to express concurrency by means of a data-parallel `for`-loop construct.

Even though both languages have built-in support for data-parallel operations, they do not facilitate truly data-parallel programming. In both languages, the composition of data-parallel operations is severely limited. In particular, data-parallel operations may not be nested, i.e., a data-parallel operation may not itself contain data-parallel operations. This limits the applicability of data-parallel operations in the setting of more irregular structures like nested arrays, sparse arrays or nested lists.

This limitation has partly been removed in later data-parallel languages like NESL [3], SAC [4], NEPAL [5] and DPH [6]. All four allow the programmer to specify nested data-parallel operations under certain constraints. However, during compilation, the nesting is removed using a flattening transformation [7–10] that lifts the inner, nested operations to the outer level. At runtime, only flat data-parallel operations are performed. Where no flattening is possible, the nested data-parallelism is ignored and sequential execution is used.

Although the trick of flattening nested data-parallelism helps in many situations, it is not a general solution. If flattening is not possible, valuable concurrency is lost. So why is nested data-parallelism not simply mapped to a nested concurrent execution at runtime?

The answer to this question is architecture dependent. On classical SIMD vector machines that were the initial target for data-parallel languages, nested data-parallelism was simply not possible. Such machines were limited to simple operations on vectors of scalar data.

In the setting of SMP and multi-core systems with their thread based programming model, nested data-parallel operations can, in theory, be expressed. However, in practice, such concurrent execution seldom pays off. Creating and synchronizing threads is a rather expensive operation. It therefore is prohibitive to use one thread per single computation. Instead, usually multiple computations of one data-parallel operation are scheduled to form one thread. In the setting of flat data-parallel operations, most of this scheduling can be performed statically. However, with nested data-parallelism, efficient static scheduling becomes impossible. Scheduling computations to threads at runtime, however, involves a major overhead. Thus, the gain from additional concurrency in many cases is overshadowed by the cost for managing threads.

The novel Microgrid architecture developed at the University of Amsterdam is about to change the rules of the game [11]. Other than existing multi-core designs, in a Microgrid threads are entirely managed by the hardware. Furthermore, threads in a Microgrid are considerably more lightweight than the classical pthreads as found on today's multi-core machines: Thread creation and synchronisation is cheap and a single core may support up to 1000 active threads at any one time and a conceptually unlimited number of waiting threads given sufficient resources.

Having cheap threads in abundance lead us to the idea to map each single computation of a data-parallel operation to its own thread. Thus, scheduling becomes straight forward. Furthermore, as the Microgrid allows for nested threads, in which a thread may spawn off new threads, such a one-to-one mapping allows us to express nested data-parallel operations as just that, nested parallelism, at hardware level, as well.

In this paper, we report on first experiences we have made with the above approach. We have devised a compilation scheme for one of SaC's data-parallel constructs that targets the systems-programming language of the Microgrid architecture. A corresponding prototypical implementation is available as part of our research compiler `sac2c`[3]. To give a first idea of the effectiveness of our approach, we have evaluated it using one of the kernels of the Livermore loop suite.

The remainder of this paper is structured as follows. In the first three sections, we give a short introduction to SaC, the Microgrid architecture and its systems programming language μTC. Then, in Section 5, we present a compilation scheme for one of SaC's data-parallel operations. Early empirical results are given in Section 6 before we close with some conclusions and future work.

2 Single Assignment C

As the name suggests, SaC is a functional subset of C, extended by multi-dimensional arrays as first class citizens. SaC has adopted as much of the syntax of C as possible to ease adaptation for programmers with a background in imperative programming. Despite its C-like appearance, the semantics of SaC code is defined by context-free substitution of expressions. "Imperative" language features like assignment chains, branches, or loops are semantically explained and internally represented as nested `let`-expressions, conditional expressions, and tail-end recursive functions, respectively. Nevertheless, wherever SaC code syntactically coincides with C code, the functional semantics of SaC also coincides with the imperative semantics of C. As a consequence, programmers may keep their preferred model of thinking while the SaC compiler may exploit the functional semantics for advanced optimisation [12].

In contrast to other array languages, SaC provides only a very small set of built-in operations on arrays: primitives to retrieve data pertaining to the

[3] `sac2c` is available at http://www.apple-core.info

structure and contents of arrays, *e.g.*, an array's rank (`dim(`*array*`)`), its shape (`shape(`*array*`)`), or individual elements (*array*`[`*index-vector*`]`). Aggregate array operations are specified in SaC itself using powerful array comprehensions, called WITH-loops. By design, these WITH-loops are data-parallel operations. In the context of this paper, we will focus on the data-parallel map operation of SaC, the `genarray` WITH-loop. Its syntax is defined in Fig. 1.

$$
\begin{array}{lll}
\mathit{Expr} & \Rightarrow & \ldots \\
 & \mid & \textbf{with}\ \{\ \lceil\, \mathit{Generator} : \mathit{Expr}\ ;\ \rfloor^{+}\ \}\ :\ \mathit{Operation} \\
\mathit{Generator} & \Rightarrow & (\ \mathit{Expr}\ \texttt{<=}\ \mathit{Identifier}\ \texttt{<}\ \mathit{Expr} \lceil\, \mathit{Filter}\, \rfloor\ ) \\
\mathit{Filter} & \Rightarrow & \textbf{step}\ \mathit{Expr}\ \lceil\ \textbf{width}\ \mathit{Expr}\ \rfloor \\
\mathit{Operation} & \Rightarrow & \textbf{genarray}\ \ (\ \mathit{Expr}\ \lceil\ ,\ \mathit{Expr}\ \rfloor\ )
\end{array}
$$

Fig. 1. Syntax of with-loop expressions

A WITH-loop is a complex expression: following the key word `with`, a non-empty list of *generator*–expression pairs defines a mapping from indices to values while the subsequent *operation* determines the overall meaning of the WITH-loop. In this paper, we only consider the `genarray` operation, which is expressed syntactically as `genarray(` ***shp, default*** `)` and creates a new array of shape ***shp***.

Each generator defines a set of indices, more precisely index vectors, along with an index variable representing elements of this set. Two expressions, which must evaluate to integer vectors of equal length, define lower and upper bounds of a rectangular index vector range. For each element of this set of index vectors the associated expression is evaluated. Depending on the variant of WITH-loop, the resulting value either defines the corresponding element value of the array to be created (`genarray`) or it is given as an argument to the fold operation (`fold`). In the case of a `genarray`-WITH-loop, elements of the result array that are not covered by the generator are initialised by the (optional) default expression in the operation part. For example, the WITH-loop

```
with {
  ([1,1] <= iv < [3,4]) : iv[0] + iv[1];
}: genarray( [3,5], 0)
```

yields the matrix $\begin{pmatrix} 0&0&0&0&0\\ 0&2&3&4&0\\ 0&3&4&5&0 \end{pmatrix}$. The generator in this example WITH-loop defines the set of 2-element vectors in the range between `[1,1]` and `[3,4]`. The index variable `iv` represents elements from this set (*i.e.*, 2-element vectors) in the associated expression `iv[0] + iv[1]`. Therefore, we compute each element of the result array as the sum of the two components of the index vector, whereas the remaining elements are initialised with the value of the default expression.

Multiple generator–expression pairs allow us to map different index sets to entirely different expressions. As a simple example, the WITH-loop

```
with {
  ([0,0] <= iv < [1,4]) : 0;
  ([0,0] <= iv < [3,1]) : 1;
  ([1,1] <= iv < [3,4]) : iv[0] + iv[1];
}: genarray( [3,5], 0)
```

yields the matrix $\begin{pmatrix} 1&0&0&0&0\\ 1&2&3&4&0\\ 1&3&4&5&0 \end{pmatrix}$. In case the generators define index sets that are not pairwise disjoint, their (textual) sequence matters: the last mapping from the list is taken.

An optional filter may be used to further restrict generators to periodic grid-like patterns, *e.g.*,

```
with {
  ([1,1] <= iv < [3,8] step [1,3] width [1,2]) : 1;
}: genarray( [3,10], 0)
```

yields the matrix $\begin{pmatrix} 0&0&0&0&0&0&0&0&0&0\\ 0&1&1&0&1&1&0&1&0&0\\ 0&1&1&0&1&1&0&1&0&0 \end{pmatrix}$. An elidet width specification defaults to 1 in each dimension; an elidet step specification likewise defaults to 1, *i.e.*, no stepping whatsoever. A width specification that equals or exceeds the step in some dimension is equivalent to a dense generator. It is noteworthy that lower and upper bounds as well as step and width specifications are fully-fledged expressions and not restricted to constants as in our illustrative examples.

More formal descriptions of WITH-loops along with detailed introductions to SAC and its programming methodology can be found in [13–15].

3 Microgrids of Self-Adaptive Virtual Processors

The key idea of the Microgrid architecture is to create a common framework for standard processing cores which have been extended in their ISAs so that they can be used concurrently in a cooperative fashion. As a unifying concept, all such cores need to adhere to the model of a Self-Adaptive Virtual Processor (SVP).

In a Microgrid, the unit of work is a family of homogeneous and indexed threads, called *microthreads*. Families of microthreads are created at *places*. A place is an abstraction of a set of processing resources configured into a ring network somewhere on chip. Families of threads are usually created on the local place, *i.e.*, on the place where the create is initiated. However, all places also do accept remote SVP actions to create families of threads from other processors. This is called a *delegation*. Delegation is a mechanism for coarse-grain distribution of work in a Microgrid. The place accepts the create parameters, distributes and executes the threads that make up the family and responds to any SVP

control actions for that family. In the conventional setting, this is equivalent to the remote execution of a unit of work (a job) on a set of processors. For more information about the SVP model see [16] and for more information about Microgrids see [11].

The threads in a family are automatically distributed to the cluster of processors but not necessarily all at the same time. If the number of threads exceeds the resources available there, thread creation may be delayed on those resources. The distribution is, however, completely deterministic and built into the implementation of the create instruction in the DRISC processors comprising the cluster. The distribution is primarily parameterised by

1. the number of threads in a family (N),
2. the number of processors in the place (P) and
3. the distribution strategy which is either local or default.

A local distribution creates all threads in one processor whereas a default distribution tries to involve all processors of a place evenly. However, this rule may be overridden whenever a paucity of resources occurs on a processor.

As the programming model for Microgrids is recursive, threads that have been distributed can themselves create subordinate families. For example, a 2-dimensional data-parallel operation could be implemented as a family of threads, each of which creates a subordinate family. Alternatively, we could create a single flat family of threads, one per element. With the former choice a pair of indices is generated automatically, whereas in the latter case we have a single index. For a nested create, each such subordinate family is distributed relative to its own parent's location in the ring. Note that there is no theoretical limit to the depth of recursion. However, a practical limit exists based on the resources used by existing threads on a processor.

It should also be noted that in order to make global identification of a distributed family feasible, we have implemented a sequentialisation of creates over the cluster's ring network. Only one thread at a time may acquire a token in the ring allowing it to create a family at a central registration place referred to as default place. Local creates can be executed concurrently across processors.

4 Programming Microgrids through μTC

The programming language μTC is an extension of standard C. In essence, it adds language support for thread creation and thread management. For the context of this paper, it suffices to introduce the two most important language constructs of μTC: `create` and `sync`. Their syntax is summarised in Figure 2. A `create`-statement of the form

```
create(fid; start; limit; step; block; location; timer)
                        statement-block
```

creates a new family of threads, where each thread executes the statement block *statement-block*. The various parameters of the `create`-statement have the following meaning:

Statememt ⇒ ...
| **create** (*Id* ; *Expr* ; *Expr* ; *Expr* ; *Expr* ; *Location* ; *Expr*)
Block
| **sync** (*Id*)

Location ⇒ (**local** | **default**)

Fig. 2. Syntax of μTC extensions

fid is an integer variable provided by the creating context; it receives the unique family identifier, which is needed for subsequent synchronisation or termination of the family.

start is an integer expression that defines the start of the index sequence for the family of threads (default value: 0).

limit is an integer expression that defines the limit of the index sequence for the family of threads (default value: unlimited).

step is an integer expression that defines the increment between index values (default value: 1).

block is an integer expression that defines the maximum number of threads allocated per processor in a single allocation round (default value: system defined).

location defines the resource on which the family will execute. A special resource called `local` forces execution of this family on the same processor as the creating environment while the `default` resource delegates the scheduling of work onto resources to the system (default value: system defined).

timer is an integer expression that restricts the number of allocation rounds to at most one per tick of a clock (default value: threads created as resources become available, subject to the constraints imposed by block).

All parameter expressions are evaluated exactly once upon execution of the `create`-statement. Any parameter except for the family identifier may be left out in favour of the default value as defined above.

Complementary to the `create`-statement, a `sync`-statement of the form

`sync(`*`fid`*`)`

synchronises the family of threads identified by the variable *`fid`*. Execution of the thread that issues the `sync`-statement is delayed until all members of the given family of threads have completed.

In addition to the `create` and `sync` statements μTC features two type qualifiers that can be used within the statement block of a thread: `index` and `shared`. An "index" variable must be of type `int`; it provides access to the thread's index within a family. "Shared" variables can be of any type; they realise a data flow style dependency chain through the family of threads. More precisely, the family member with the `start`-index takes the value of a shared variable from the creating context, where a variable of the same name must exist. All threads block upon reading a shared variable until its immediate predecessor or, in case

of the thread associated to the `start`-index, the creating thread performs its first write operation on the shared variable. After the family has been synchronised using the `sync`-statement the value written by the thread with the greatest index becomes available in the creating context.

```
...
int fid, sum=0;
create (fid; 0; n; 1; ; ;) {
  index int i;
  shared int sum;
  sum = sum + V[i];
}
sync(fid);
... sum ...
```

Fig. 3. μTC example code fragment for a reduction operation.

Fig. 3 illustrates the interaction of `create` and `sync` statements with `index` and `shared` type qualifiers through the example of a simple reduction operation on some previously defined vector `V` of size `n`. We create one thread per element of `V`. While the index variable `i` provides access to the local thread ID for indexing into the vector `V`, the shared variable `sum` establishes a dependency chain that guarantees a deterministic left-to-right sequence of additions, despite the concurrent execution of the threads. In practice, each thread starts by fetching the corresponding value of `V` from memory and then blocks on the availability of the reduction variable `sum`. At first glance, this sequentialises the reduction operation, but in fact each thread loads the necessary data from memory fully concurrently. Only the final reductions are sequentialised, thus ensuring deterministic program behaviour. Following the `sync`-statement the creating context may safely refer to the value of `sum` that holds the reduction result, *i.e.*, the sum of all elements of `V`.

5 Compiling SaC to μTC

All data-parallelism in SaC is expressed in terms of the multi-generator WITH-loops as introduced in Section 2. During the compilation process of SaC programs, these are transformed by means of several optimisations (see [14] for details). These transformations try to avoid the creation of arrays that hold intermediate results and, more importantly, they ensure that the resulting multi-generator WITH-loops have non-overlapping generators. This property together with the side-effect free nature of SaC guarantees that all index-vector sets in any given multi-generator WITH-loop can be traversed in arbitrary order with-

out affecting the overall result, i.e., they can be translated directly into `create` instructions of μTC.

As discussed in the previous two sections, the Microgrid architecture allows for a large number of active threads and facilitates nested concurrency by means of nested `create` operations. This allows us to compile each generator into a nesting of `create` instructions, where each dimension of the generator leads to one `create` instruction. Only if the `width` option is being used, we may actually create two nested `create` instructions per dimension. In either case, each element of the result of a WITH-loop is computed by its own thread.

Figure 4 shows a formalisation of the basic compilation scheme for multi-generator `genarray`-WITH-loops. It consists of rules of the form $\mathcal{C}[\![\mathcal{D},\ expr]\!] =$

$$\mathcal{C}\left[\!\!\left[\mathcal{D},\ \begin{array}{l}\texttt{a = with \{}\\ \quad\texttt{(l}_1\texttt{ <= iv <= u}_1\texttt{ step s}_1\texttt{ width w}_1\texttt{)}\quad\texttt{: Op}_1\texttt{(iv);}\\ \quad\vdots\qquad\qquad\qquad\qquad\qquad\vdots\\ \quad\texttt{(l}_m\texttt{ <= iv <= u}_m\texttt{ step s}_m\texttt{ width w}_m\texttt{) : Op}_m\texttt{(iv);}\\ \texttt{\} : genarray(}\ shp\texttt{);}\end{array}\right]\!\!\right] \tag{1}$$

$$= \begin{cases}\texttt{a = MALLOC(}\ shp\texttt{);}\\ \mathcal{C}[\texttt{global, (l}_1\texttt{ <= iv <= u}_1\texttt{ step s}_1\texttt{ width w}_1\texttt{) : a[iv] = Op}_1\texttt{(iv)}]\\ \vdots\\ \mathcal{C}[\texttt{global, (l}_m\texttt{ <= iv <= u}_m\texttt{ step s}_m\texttt{ width w}_m\texttt{) : a[iv] = Op}_m\texttt{(iv)}]\end{cases}$$

$$\mathcal{C}\left[\!\!\left[\mathcal{D},\ \begin{array}{l}\texttt{([}l_i\texttt{...}l_{n-1}\texttt{] <= [}iv_i\texttt{...}iv_{n-1}\texttt{] <= [}u_i\texttt{...}u_{n-1}\texttt{]}\\ \quad\texttt{step [}s_i\texttt{...}s_{n-1}\texttt{] width [}w_i\texttt{...}w_{n-1}\texttt{]) : Ass}\end{array}\right]\!\!\right] \tag{2}$$

$$= \begin{cases}\texttt{\{}\\ \quad\texttt{int fid;}\\ \quad\texttt{create(fid, }l_i\texttt{; }u_i\texttt{; }s_i\texttt{;;}\mathcal{D}\texttt{;) \{}\\ \quad\quad\texttt{index int }iv_i\texttt{;}\\ \quad\quad\texttt{int stop = MIN(}\ iv_i\texttt{+}w_i\texttt{-1, }u_i\texttt{);}\\ \quad\quad\texttt{int fid;}\\ \quad\quad\texttt{create(fid; }iv_i\texttt{, stop; 1;;local;) \{}\\ \quad\quad\quad\texttt{index int }iv_i\texttt{;}\\ \quad\quad\quad\mathcal{C}\left[\!\!\left[\texttt{local},\ \begin{array}{l}\texttt{([}l_{i+1}\texttt{...}l_{n-1}\texttt{] <= [}iv_{i+1}\texttt{...}iv_{n-1}\texttt{] <= [}u_{i+1}\texttt{...}u_{n-1}\texttt{]}\\ \quad\texttt{step [}s_{i+1}\texttt{...}s_{n-1}\texttt{] width [}w_{i+1}\texttt{...}w_{n-1}\texttt{]) : Ass}\end{array}\right]\!\!\right]\\ \quad\quad\texttt{\}}\\ \quad\quad\texttt{sync(fid);}\\ \quad\texttt{\}}\\ \quad\texttt{sync(fid);}\\ \texttt{\}}\end{cases}$$

$$\mathcal{C}[\![\mathcal{D},\ \texttt{([] <= iv <= [] step [] width []) : Ass}\ ]\!] = \texttt{Ass;} \tag{3}$$

Fig. 4. Compilation of multi generator `genarray`-WITH-loops.

expr′, which denote context-free substitutions of SAC program fragments *expr* by μTC program fragments *expr′*. We use the argument $\mathcal{D}$ to distinguish between the two distribution modes `global` and `local`. The distribution mode `global` triggers the distribution of the computation of the current level among all cores in a Microgrid. If the distribution `local` is used, the following level will be

computed on the current processing core only. Previous experiments have shown that distributing the threads at all levels of a nested create globally can have an adverse effect on runtime performance [17]. We therefore have chosen to only distribute the threads created by the outermost `create` operation globally. $\mathcal{D}$ can initially be set to any value.

Rule (1) allocates the memory for the result using `a = MALLOC(` *shp*`);` and it triggers the successive compilation of the individual generators. An explicit initialization of the result array is not required as the generator sets are guaranteed to be a partition of all legal index vectors. In the applications of the compilation scheme to the individual generators, the expressions to be evaluated are transformed into assignments of the form `a[iv] = Op( iv)`, which ensures correct insertion of the computed values into the result array. Furthermore, as we transform the outermost dimension next, we set the distribution to `global`. Thus, the outermost dimension of every WITH-loop is distributed across the Microgrid.

The last two rules concern the compilation of generator expressions into a nesting of `create`-instructions. As shown in rule (2), for each component of the indexing vector `iv`, two nested `create`-instructions are created: An outer `create` which creates threads using the lower bound l_i, upper bound u_i and the step s_i as thread indices. Hence, the index of that create can directly be utilised as index component iv_i. The inner `create` is used for treating width components larger than `1`. Note here that the inner `create`, as well as the subsequent `sync` can safely be omitted whenever the width component under consideration is 1. The body of the inner `create` derives from recursively applying the compilation scheme to the generator with its leading index vector components being eliminated. We use the `local` distribution scheme during the recursive descend and for inner `create` operations. This ensures that only the outermost `create` instruction of a WITH-loop is globally distributed.

Rule (3) covers the creation of the innermost body. It simply replaces the empty generator by the assignment associated to it.

Since we use nested `create` operations to spawn the threads for the data-parallel computation, we have to synchronize on the results on each level. However, as all threads of a WITH-loop are independent, it suffices to synchronize on whole families of threads. We implement this barrier synchronisation by inserting appropriate `sync` statements after each `create` operation in rule (2).

A compilation scheme for entire SaC programs is beyond the scope of this paper and would not provide any further insights into the code generation for WITH-loops.

6 Performance Evaluation

We have extended our research compiler `sac2c` with a prototypical backend for the Microgrid architecture using the compilation scheme presented in the previous section. For the experiments discussed in this section, we have used revision 16308 of `sac2c`. The resulting μTC code was then compiled using build 2668-2661 of the mgsim-slc toolchain. To obtain runtimes, we have used the

cycle accurate MGSim2 emulator for the Microgrid architecture. The emulator was configured to emulate a banked memory system and varying numbers of processing cores.

For our early evaluations, we have chosen the first Livermore loop kernel. This kernel belongs into the category 'embarrassingly parallel' and was small enough to be run on emulated hardware. We have used a comparatively small problem size of 995 data elements to keep emulation runs sufficiently fast.

To investigate whether our approach yields competitive runtimes on the Microgrid architecture, we have furthermore evaluated the Microgrid reference implementation of the first Livermore loop. The setup of the emulator and the problem size were identical in both measurements.

The actual runtimes were measured by instrumenting the emulator. We have recorded the cycle counter just before and after the code that computes the actual loop. This allows us to factor out the runtime cost for setup and tear down of the different runtime systems used by the SAC implementation and the hand-coded μTC version. In particular, the SAC version uses dynamic memory allocation whereas the hand-coded version uses static, pre-allocated memory regions. However, the actual loop bodies were not modified.

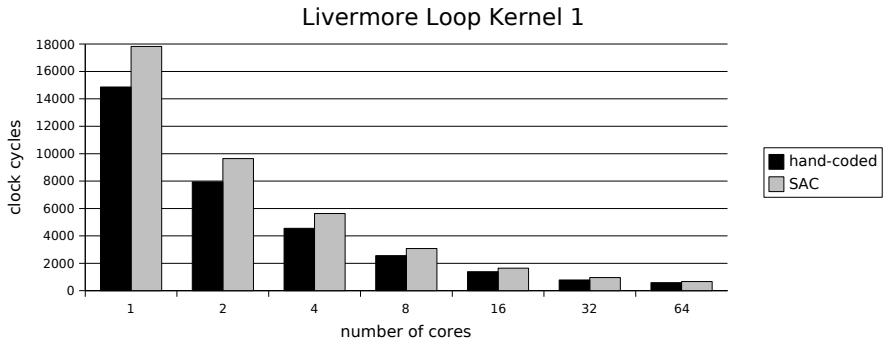

Fig. 5. Number of clock cycles required to compute one iteration of the Livermore loop kernel 1 using a varying number of cores.

Figure 5 shows the results of our experiment. As can be seen, the hand-coded μTC version slightly outperforms the corresponding SAC implementation regardless of the number of cores used. Nonetheless, both version scale equally well with increasing number of cores. We have plotted the corresponding speedups in Figure 6 to better visualise the scaling behaviour of the two implementations. Note here that we use logarithmic scales for both the x-axis (number of cores) and the y-axis (speedup).

As can be seen, both implementations have nearly the same scaling behaviour. They scale linearly up to 32 cores. For more than 32 cores, the speedup grows more slowly. We explain this with the relatively small problem size: Be-

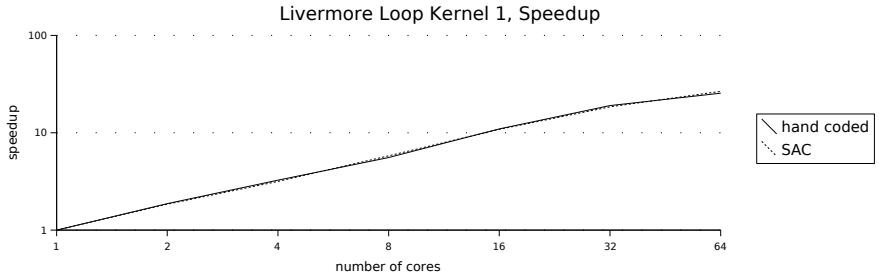

Fig. 6. Relative speedup when computing one iteration of the Livermore loop kernel 1 using a varying number of cores.

yond a certain amount of cores, the amount of available computation simply does not suffice to keep all cores under full load.

7 Conclusions and Future Work

This paper presents early results from evaluating the implementation of truly nested data-parallelism using the Microgrid architecture. Other than existing approaches, our implementation of nested data-parallelism is true as we do not flatten nested concurrency into flat operations. Instead, we directly map the concurrency that arises from nested data-parallel operations in SaC to the executing machinery.

To this effect, we have devised a compilation scheme that translates the `genarray` WITH-loop of SaC to nested parallel loops. As our runtime evaluation shows, such a direct approach yields competitive runtimes for the first Livermore loop kernel.

We are still in the early stages of exploring the Microgrid architecture as a target platform for SaC. In particular, our compiler is currently limited to simple cases like the embarrassingly parallel subset of the Livermore loops. For the full version of this paper, we hope to be able to show runtimes for more complex nested algorithms like Barnes-Hut n-body or sparse-matrix vector products.

Acknowledgements

We would like to thank Raphael Poss and Michael Hicks from the University of Amsterdam for providing a reference implementation of the livermore loop kernels in μTC and corresponding runtimes on the Microgrid architecture.

References

1. High Performance Fortran Forum: High Performance Fortran language specification V1.1. (1994)

2. Feo, J.: SISAL. Technical Report UCRL-JC-110915, Lawrence Livermore National Laboratory, LLNL, Livermore California (1992)
3. Blelloch, G.: NESL: A Nested Data-Parallel Language (Ver sion 3.0). Carnegie Mellon University. (1994)
4. Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming **15** (2005) 353–401
5. Chakravarty, M.M., Lechtchinsky, R., Keller, G., Pfannenstiel, W., Informatik, F., Informatik, F.: Nepal - nested data-parallelism in haskell. In: In Euro-Par 01, Springer-Verlag (2001) 524–534
6. Chakravarty, M.M.T., Leshchinskiy, R., Jones, S.P., Keller, G., Marlow, S.: Data parallel haskell: a status report. In: DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming, New York, NY, USA, ACM Press (2007) 10–18
7. Blelloch, G.E., Sabot, G.W.: Compiling collection-oriented languages onto massively parallel computers. J. Parallel Distrib. Comput. **8** (1990) 119–134
8. Chakravarty, M.M.T., Keller, G.: More types for nested data parallel programming. In: ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM (2000) 94–105
9. Leshchinskiy, R., Chakravarty, M.M.T., Keller, G.: Higher order flattening. In: In Third International Workshop on Practical Aspects of High-Level Parallel Programming (PAPP 2006), Springer-Verlag (2006) 920–928
10. Grelck, C., Scholz, S.B., Trojahner, K.: WITH-Loop Scalarization – Merging Nested Array Operations. In Michaelson, G., Trinder, P., eds.: Proc. of the 15th International Workshop on Implementation of Functional Languages (IFL'03), Edinburgh, UK, Selected Papers. Volume 3145 of LNCS., Springer (2004) 118–134
11. Bernard, T., Bousias, K., Guang, L., Jesshope, C., Lankamp, M., van Tol, M., Zhang, L.: A general model of concurrency and its implementation as many-core dynamic RISC processors. In: Embedded Computer Systems: Architectures, Modeling, and Simulation, 2008. SAMOS 2008. International Conference on, IEEE (2008) 1–9
12. Grelck, C., Scholz, S.B.: Merging compositions of array skeletons in SAC. Journal of Parallel Computing **32** (2006) 507–522
13. Scholz, S.B.: Single Assignment C — efficient support for high-level array operations in a functional setting. Journal of Functional Programming **13** (2003) 1005–1059
14. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. International Journal of Parallel Programming **34** (2006) 383–427
15. Grelck, C., Scholz, S.B.: Sac: off-the-shelf support for data-parallelism on multicores. In: DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming, New York, NY, USA, ACM (2007) 25–33
16. Jesshope, C.: A model for the design and programming of multi-cores. In Grandinetti, L., ed.: High performance Computing and Grids in Action. Volume 16 of Advances in Parallel Computing., IOS Press (2008) 37–55
17. Grelck, C., Herhut, S., Jesshope, C., Joslin, C., Lankamp, M., Scholz, S.B., Shafarenko, A.: Compiling the Functional Data-Parallel Language SaC for Microgrids of Self-Adaptive Virtual Processors. In: 14th Workshop on Compilers for Parallel Computing (CPC'09), IBM Research Center, Zurich, Switzerland, accepted for publication (2009)